

REPORT 5F6AC62264B93A00182349E1

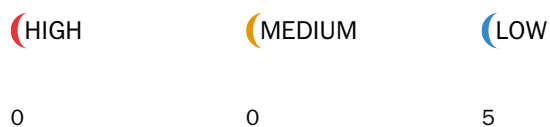
Created	Wed Sep 23 2020 03:50:58 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	kohshi.shiba@gmail.com

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
ab727016-2a79-4ed9-9a3b-b91d6a43d871	/contracts/linerv1.sol	5

Started	Wed Sep 23 2020 03:51:02 GMT+0000 (Coordinated Universal Time)
Finished	Wed Sep 23 2020 04:36:22 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Mythx-Vscode-Extension
Main Source File	/Contracts/Linerv1.Sol

DETECTED VULNERABILITIES



ISSUES

- LOW** A floating pragma is set.
The current pragma Solidity directive is `""^0.6.0""`. It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.
- SWC-103**

Source file
/contracts/linerv1.sol
Locations

```
5 | */  
6 |  
7 | pragma solidity ^0.6.0;  
8 | pragma experimental ABIEncoderV2;
```

- LOW** An assertion violation was triggered.
It is possible to cause an assertion violation. Note that Solidity `assert()` statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use `require()` instead of `assert()` if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).
- SWC-110**

Source file
/contracts/linerv1.sol
Locations

```
487 | tokenValue = tokenValue.sqrt();  
488 | tokenValue -= initialPrice;  
489 | tokenValue /= priceIncrement;  
490 | tokenValue -= supply;  
491 | if (
```

LOW

A control flow decision is made based on The block.timestamp environment variable.

SWC-116

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

/contracts/linerv1.sol

Locations

```
130 | ///@dev a modifier manages market transitions
131 | modifier marketStatusTransitions() {
132 |     if (marketStatus == MarketStatus.BeforeTrading && now >= startTime)
133 |         _nextMarketStatus();
134 |
135 |     if (marketStatus == MarketStatus.Trading && now >= endTime) {
136 |         _nextMarketStatus();
```

LOW

A control flow decision is made based on The block.timestamp environment variable.

SWC-116

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

/contracts/linerv1.sol

Locations

```
133 |     _nextMarketStatus();
134 | }
135 | if (marketStatus == MarketStatus.Trading && now >= endTime)
136 |     _nextMarketStatus();
137 |
138 | _;
139 | }
```

LOW Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

SWC-123

Source file

/contracts/linerv1.sol

Locations

```
17 import "./IMarket.sol";
18
19 contract Linerv1 is ERC1155, IMarket {
20     using SafeMath for uint256;
21     using Sqrt for uint256;
22     using Address for address;
23
24     /**
25      * EVENTS
26      */
27
28     event Buy(
29         address buyer,
30         address to,
31         uint256 outcomeIndex,
32         uint256 investValue,
33         uint256 returnValue
34     );
35     event Sell(
36         address seller,
37         address to,
38         uint256 outcomeIndex,
39         uint256 sellValue,
40         uint256 returnValue
41     );
42     event Claimed(
43         address owner,
44         uint256 outcomeIndex,
45         uint256 claimedValue,
46         uint256 returnValue
47     );
48     event FeeCollected(
49         address beneficiary,
50         uint256 totalAccrued,
51         uint256 collected
52     );
53     event FeeWithdrawal(address beneficiary, uint256 amount);
54     event MarketSettled(uint256[] report, uint256[] payout);
55     event MarketStatusChanged(MarketStatus statusValue);
56
57     /**
58      * MARKET CONSTANTS
59      */
60
61     /// @dev Global denominator e.g., 1.000% = 1000 & need to be divided by 100000
62     uint256 private constant GLOBAL_DENOMINATOR = 100000;
63
64     /// @dev The factory address that deployed this contract
65     address private factory;
66
67     /// @dev True once initialized through initialize()
68     bool private initialized;
69
70     /// @dev Decimals for option tokens
71     uint8 public decimals = 18;
```

```

72
73 /// @dev The price to buy option increase as new token issued
74 uint256 public priceIncrement;
75
76 /// @dev The minimum amount of 'currency' investment accepted.
77 uint256 public minInvestment;
78
79 /// @dev The minimum amount of 'currency' investment accepted.
80 uint256 public startPrice;
81
82 /// @dev When the sell option is disabled, option tokens cannot be sold. (0 = true)
83 uint256 public sellOption;
84
85 ///@dev Market contents (Registered when market is created)
86 bytes32 public hashID;
87 uint256 public startTime;
88 uint256 public endTime;
89 uint256 public reportTime;
90 address public oracle;
91 IERC20 public token;
92 address[] public beneficiaries;
93 uint256[] public shares;
94
95 /**
96  * OUTCOMES
97  */
98 struct Outcome {
99     uint256 supply;
100     uint256 reserve;
101     uint256 dividend;
102 }
103 ///@dev mapping for each outcome
104 mapping(uint256 => Outcome) public outcome;
105
106 ///@dev total number of outcomes
107 uint256 public outcomeNumbers;
108
109 /**
110  * MARKET VARIABLES
111  */
112
113 /// @dev collected fee balances
114 mapping(address => uint256) public collectedFees;
115
116 /**
117  * MARKET STATES MANAGEMENT
118  */
119
120 ///@dev Market status transition management
121 enum MarketStatus {BeforeTrading, Trading, Reporting, Finalized}
122 MarketStatus public marketStatus;
123
124 ///@dev a modifier checks the current market status
125 modifier atMarketStatus(MarketStatus _marketStatus) {
126     require(marketStatus == _marketStatus);
127 }
128
129
130 ///@dev a modifier manages market transitions
131 modifier marketStatusTransitions() {
132     if (marketStatus == MarketStatus.BeforeTrading && now >= startTime) {
133         nextMarketStatus();
134     }

```

```

135 if (marketStatus == MarketStatus.Trading && now >= endTime) {
136     nextMarketStatus();
137 }
138 ==
139 }
140
141 constructor() public {
142
143     /**
144      * @dev Validate market
145      * This function validates the argument set for initialization.
146      * Can be called before contract deployments.
147      * question = the thesis of the prediction market
148      * outcomes = potential outcomes
149      * conditions[0] = starttime
150      * conditions[1] = endTime
151      * conditions[2] = reportTime
152      * conditions[3] = priceIncrement
153      * conditions[4] = sell option (0:yes 1:no)
154      * conditions[5] = minimum investment value
155      * conditions[6] = start price
156      * references[0] = ERC20 token used as the collateral
157      * references[1] = oracle address settles the market
158      * beneficiaries[] = beneficiary addresses collect fees
159      * shares[] = fee shares
160      * detail = any additional info about the market
161      */
162     function validate(
163         string memory _settings,
164         uint256 _outcomeNum,
165         uint256[] memory _conditions,
166         address[] memory _references,
167         address[] memory _beneficiaries,
168         uint256[] memory _shares
169     ) public override view returns (bool) {
170         require(bytes(_settings).length > 10);
171         require(_outcomeNum >= 2 && _outcomeNum <= 10);
172         require(
173             _conditions[1].sub(_conditions[0]) > 1 days &&
174             _conditions[1].sub(now) > 1 days &&
175             _conditions[2] >= _conditions[1]
176         );
177         require(_conditions[3] > 0);
178         require(_conditions[4] < 2);
179         require(_references[0] != address(0) && _references[1] != address(0));
180         require(_beneficiaries.length <= 3);
181         require(_beneficiaries.length == _shares.length);
182
183         uint256 share;
184         for (uint256 i = 0; i < _shares.length; i++) {
185             share = share.add(_shares[i]);
186             require(GLOBAL_DENOMINATOR > share);
187
188             return true;
189         }
190
191         /**
192          * @dev Initialize market
193          * This function registers market conditions.
194          * arguments are verified by the 'validate' function.
195          */
196         function initialize(
197             string memory _settings,

```

```

198     uint256 _outcomeNum;
199     uint256[] memory _conditions;
200     address[] memory _references;
201     address[] memory _beneficiaries;
202     uint256[] memory _shares;
203     public override returns (bool) {
204         require(
205             validate(
206                 _settings,
207                 _outcomeNum,
208                 _conditions,
209                 _references,
210                 _beneficiaries,
211                 _shares
212             )
213         );
214
215         require(initialized == false);
216         initialized = true;
217         outcomeNumbers = _outcomeNum;
218         startTime = _conditions[0];
219         endTime = _conditions[1];
220         reportTime = _conditions[2];
221         priceIncrement = _conditions[3];
222         sellOption = _conditions[4];
223         minInvestment = _conditions[5];
224         startPrice = _conditions[6];
225         token = IERC20(_references[0]);
226         oracle = _references[1];
227         beneficiaries = _beneficiaries;
228         shares = _shares;
229         factory = msg.sender;
230
231         hashID = keccak256(
232             abi.encodePacked(
233                 _settings,
234                 _outcomeNum,
235                 _conditions,
236                 _references,
237                 _beneficiaries,
238                 _shares
239             )
240         );
241
242         marketStatus = MarketStatus.BeforeTrading;
243         return true;
244     }
245
246     /**
247     * @dev Buy
248     * Market participants can buy option tokens through this function.
249     * _params[0] investmentAmount,
250     * _params[1] minTokensBought,
251     * _params[2] outcomeIndex,
252     * _params[3] fee,
253     * _addresses[0] owner,
254     * _addresses[1] to
255     * _addresses[2] beneficiary,
256     */
257     function buy(uint256[] memory _params, address[] memory _addresses)
258     public
259     marketStatusTransitions
260     atMarketStatus(MarketStatus.Trading)

```

```

261 |
262 | require(_params[1] > 0, "MUST_BUY_AT_LEAST_1");
263 |
264 | // Calculate the tokenValue for this investment
265 | uint256 tokenValue = calcBuyAmount(_params[0], _params[2], _params[3]);
266 | require(tokenValue >= _params[1], "PRICE_SLIPPAGE");
267 |
268 | ERC20(token).transferFrom(msg.sender, address(this), _params[0]);
269 | if (shares.length > 0 || _params[3] > 0) {
270 |     uint256 afterFee = _collectFees(
271 |         _params[0],
272 |         _params[3],
273 |         addresses[2]
274 |     );
275 |     outcome[_params[2]].reserve = outcome[_params[2]].reserve.add(
276 |         afterFee
277 |     );
278 | } else {
279 |     outcome[_params[2]].reserve = outcome[_params[2]].reserve.add(
280 |         _params[0]
281 |     );
282 | }
283 |
284 | _mint(addresses[1], _params[2], tokenValue, "");
285 | outcome[_params[2]].supply = outcome[_params[2]].supply.add(tokenValue);
286 |
287 | emit Buy(msg.sender, addresses[1], _params[2], _params[0], tokenValue);
288 | }
289 |
290 | /**
291 |  * @dev Sell
292 |  * Market participants can sell option tokens through this function.
293 |  * _params[0] sellAmount,
294 |  * _params[1] minReturned,
295 |  * _params[2] outcomeIndex,
296 |  * _addresses[0] owner,
297 |  * _addresses[1] to
298 |  */
299 | function sell(uint256 memory _params, address[] memory _addresses)
300 | public
301 | marketStatusTransitions
302 | atMarketStatus(MarketStatus.Trading)
303 | {
304 |     require(
305 |         balanceOf(_addresses[0], _params[2]) >= _params[0],
306 |         "INSUFFICIENT_AMOUNT"
307 |     );
308 |     require(
309 |         msg.sender == _addresses[0] ||
310 |         _operatorApprovals[_addresses[0]](msg.sender),
311 |         "NOT_ELIGIBLE_TO_SELL"
312 |     );
313 |     uint256 returnValue = calcSellAmount(_params[0], _params[2]);
314 |     require(returnValue >= _params[1], "PRICE_SLIPPAGE");
315 |     _burn(_addresses[0], _params[2], _params[0]);
316 |     outcome[_params[2]].reserve = outcome[_params[2]].reserve.sub(
317 |         returnValue
318 |     );
319 |     outcome[_params[2]].supply = outcome[_params[2]].supply.sub(_params[0]);
320 |     ERC20(token).transfer(_addresses[1], returnValue);
321 |     emit Sell(
322 |         msg.sender,
323 |         _addresses[0],

```



```

324     _params[2],
325     _params[0],
326     returnValue
327 }
328 }
329
330 /**
331  * @dev Settle
332  * Registered oracle settles market by reporting payout shares.
333  */
334 function settle(uint256[] memory report)
335 public
336 marketStatusTransitions
337 atMarketStatus{MarketStatus.Reporting}
338 {
339     require(msg.sender == oracle, "UNAUTHORIZED_ORACLE");
340
341     uint256 total;
342     for (uint256 i = 0; i < report.length; i++)
343     {
344         total = total.add(report[i]);
345     }
346     require(
347         total == GLOBAL_DENOMINATOR && report.length == outcomeNumbers,
348         "INVALID_REPORT"
349     );
350     nextMarketStatus();
351
352     /**
353      * If there is no supply for a winning option,
354      * the dividend of that will be distributed to all token holders.
355      */
356     uint256 totalReserve;
357     uint256 totalSupply;
358     uint256 bonus;
359     uint256[] memory _payout = new uint256[](outcomeNumbers);
360     for (uint256 i = 0; i < outcomeNumbers; i++)
361     {
362         totalReserve = totalReserve.add(outcome[i].reserve);
363         totalSupply = totalSupply.add(outcome[i].supply);
364     }
365     for (uint256 i = 0; i < outcomeNumbers; i++)
366     {
367         if (outcome[i].supply == 0)
368         {
369             uint256 temp = BigDiv.bigDiv2x1(
370                 totalReserve,
371                 report[i],
372                 GLOBAL_DENOMINATOR
373             );
374             bonus = bonus.add(temp);
375         }
376     }
377     for (uint256 i = 0; i < report.length; i++)
378     {
379         if (bonus > 0)
380         {
381             if (outcome[i].supply != 0)
382             {
383                 uint256 allocation = BigDiv.bigDiv2x1(
384                     totalReserve,
385                     report[i],
386                     GLOBAL_DENOMINATOR
387                 );
388                 uint256 bonusShare = BigDiv.bigDiv2x1(
389                     bonus,
390                     outcome[i].supply,
391                     totalSupply
392                 );
393                 outcome[i].dividend = bonusShare + allocation;

```

```

387     _payout[i] = bonusShare + allocation;
388 }
389 else
390     uint256 allocation = BigDiv.bigDiv2x1;
391     totalReserve;
392     report i;
393     GLOBAL_DENOMINATOR
394 }
395 outcome[i].dividend = allocation;
396 _payout[i] = allocation;
397 }
398 }
399
400 emit MarketSettled(report, _payout);
401 }
402
403 /**
404  * @dev Winnig token holders can claim redemption through this function
405  */
406 function claim(address account)
407 public
408 atMarketStatus(MarketStatus.Finalized)
409 {
410     uint256 redemption;
411     for (uint256 i = 0; i < outcomeNumbers; i++) {
412         uint256 balance = balanceOf(account, i);
413         if (balance > 0) {
414             if (outcome[i].dividend > 0) {
415                 uint256 value = BigDiv.bigDiv2x1;
416                 outcome[i].dividend;
417                 balance;
418                 outcome[i].supply;
419             }
420             _burn(account, i, balance);
421             outcome[i].supply = outcome[i].supply.sub(balance);
422             outcome[i].dividend = outcome[i].dividend.sub(value);
423             redemption = redemption.add(value);
424             emit Claimed(account, i, balance, value);
425         }
426     }
427
428     if (redemption > 0) {
429         IERC20(token).transfer(account, redemption);
430     }
431 }
432
433 /**
434  * @dev Beneficiaries can withdraw fees through this function.
435  */
436 function withdrawFees(address account) public {
437     uint256 amount = collectedFees[account];
438     collectedFees[account] = 0;
439     emit FeeWithdrawal(account, amount);
440     IERC20(token).transfer(account, amount);
441 }
442
443 /**
444  * @dev Calc late estimate option token amount for the investment at a time.
445  */
446 function calcBuyAmount(
447     uint256 investmentAmount,
448     uint256 outcomeIndex,
449     uint256 fee

```

```

450 | public view returns (uint256) {
451 |     if (investmentAmount < minInvestment) {
452 |         return 0;
453 |     }
454 |
455 |     /**
456 |      * Calculate the fee rate for this investment.
457 |      */
458 |
459 |     uint256 afterFee;
460 |     if (shares.length > 0 || fee > 0) {
461 |         uint256 feeRate;
462 |         for (uint256 i = 0; i < shares.length; i++) {
463 |             feeRate += feeRate.add(shares[i]);
464 |         }
465 |         feeRate += feeRate.add(fee);
466 |         require(feeRate < GLOBAL_DENOMINATOR);
467 |         uint256 fees = BigDiv.bigDiv2x1(
468 |             investmentAmount,
469 |             feeRate,
470 |             GLOBAL_DENOMINATOR
471 |         );
472 |         afterFee = investmentAmount.sub(fees);
473 |     } else {
474 |         afterFee = investmentAmount;
475 |     }
476 |
477 |     /**
478 |      * Calculate the tokenValue for this investment.
479 |      */
480 |     uint256 supply = outcome.outcomeIndex.supply;
481 |     uint256 reserve = outcome.outcomeIndex.reserve;
482 |     uint256 newReserve = reserve + afterFee;
483 |     uint256 initialPrice = startPrice * 1e18;
484 |     uint256 tokenValue = ((2 *
485 |         priceIncrement.mul(1e18)).mul(newReserve.mul(1e18))) +
486 |         (initialPrice**2));
487 |     tokenValue = tokenValue.sqrt();
488 |     tokenValue -= initialPrice;
489 |     tokenValue /= priceIncrement;
490 |     tokenValue -= supply;
491 |     if (
492 |         marketStatus == MarketStatus.Trading ||
493 |         marketStatus == MarketStatus.BeforeTrading
494 |     ) {
495 |         return tokenValue;
496 |     } else {
497 |         return 0;
498 |     }
499 | }
500 |
501 | /**
502 |  * @dev Calc late estimate collateralize token value for selling ptions
503 |  */
504 | function calcSellAmount(uint256 sellAmount, uint256 outcomeIndex
505 | public
506 | view
507 | returns (uint256)
508 | {
509 |     require(sellOption == 0, "SELL_OPTION_IS_DISABLED");
510 |     if (marketStatus == MarketStatus.Trading) {
511 |         uint256 supply = outcome.outcomeIndex.supply;
512 |         uint256 reserve = outcome.outcomeIndex.reserve;

```

```

513 require(supply >= sellAmount, "BEYOND_SUPPLY");
514 if (supply == 0) {
515     return 0;
516 }
517 /**
518  * Calculate the token return for this reserve token sale.
519  */
520 uint256 supplyAfter = supply - sellAmount;
521 if (supplyAfter == 0) {
522     return reserve;
523 } else {
524     uint256 price = BigDiv
525         (bigDiv2x1(supplyAfter, priceIncrement, 1e18)
526         .add(startPrice));
527     uint256 reserveAfter = BigDiv
528         (bigDiv2x1(price.add(startPrice), supplyAfter, 1e18)
529         .div(2));
530     uint256 retVal = reserve - reserveAfter;
531     return retVal;
532 }
533 } else {
534     return 0;
535 }
536 }
537
538 /**
539  * @dev function to get pool balance for each option
540  */
541 function getStake(uint256 outcomeIndex) public view returns (uint256) {
542     return outcome[outcomeIndex].reserve;
543 }
544
545 /**
546  * @dev function to get supply for each option
547  */
548 function getSupply(uint256 outcomeIndex) public view returns (uint256) {
549     return outcome[outcomeIndex].supply;
550 }
551
552 /**
553  * @dev a function to check the factory address
554  */
555 function creator() public override view returns (address) {
556     return factory;
557 }
558
559 /**
560  * @dev Validate market question and outcome lists
561  */
562 function validateHash()
563     string memory _settings
564     uint256 _outcomeNum
565     uint256[] memory _conditions
566     address[] memory _references
567     address[] memory _beneficiaries
568     uint256[] memory _shares
569     public override view returns (bool) {
570     bytes32 hash = keccak256(
571         abi.encodePacked(
572             _settings,
573             _outcomeNum,
574             _conditions,
575             _references,

```

```

576     _beneficiaries
577     _shares
578 }
579 %
580 return _hash == hashID;
581 }
582
583 function _nextMarketStatus() internal {
584     marketStatus = MarketStatus(uint256(marketStatus) + 1);
585     emit MarketStatusChanged(marketStatus);
586 }
587
588 function _collectFees(
589     uint256 amount,
590     uint256 fee,
591     address beneficiary
592 ) internal returns (uint256) {
593     uint256 fees;
594     for (uint256 i = 0; i < _shares.length; i++) {
595         uint256 portion = BigDiv.bigDiv2x1(
596             amount,
597             _shares[i],
598             GLOBAL_DENOMINATOR
599         );
600         collectedFees[_beneficiaries[i]] = collectedFees[_beneficiaries[i]]
601             .add(portion);
602         fees = fees.add(portion);
603         emit FeeCollected(
604             _beneficiaries[i],
605             collectedFees[_beneficiaries[i]],
606             portion
607         );
608     }
609     if (fee > 0) {
610         uint256 portion = BigDiv.bigDiv2x1(amount - fee, GLOBAL_DENOMINATOR);
611         collectedFees[_beneficiary] = collectedFees[_beneficiary].add(
612             portion
613         );
614         fees = fees.add(portion);
615         emit FeeCollected(_beneficiary, collectedFees[_beneficiary], portion);
616     }
617     return amount.sub(fees);
618 }
619 %

```